HAL
open science

Testing Peers' Volatility

Eduardo de Almeida, Gerson Sunye, Yves Le Traon, Patrick Valduriez

To cite this version:

Eduardo de Almeida, Gerson Sunye, Yves Le Traon, Patrick Valduriez. Testing Peers' Volatility. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 2008, L'Aquila, Italy, Italy. inria-00371199

HAL Id: inria-00371199

https://inria.hal.science/inria-00371199v1

Submitted on 30 Mar 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Testing Peers' Volatility

Eduardo Cunha de Almeida*, Gerson Sunyé*, Yves Le Traon† and Patrick Valduriez‡
*LINA - Université de Nantes
Email: {eduardo.almeida, gerson.sunye}@univ-nantes.fr
†IT-TELECOM Bretagne,
Email: yves.letraon@telecom-bretagne.eu
‡INRIA & LINA - Nantes,
Email: patrick.valduriez@inria.fr

***Abstract*—Peer-to-peer (P2P) is becoming a key technology for software development, but still lacks integrated solutions to build trust in the final software, in terms of correctness and security. Testing such systems is difficult because of the high numbers of nodes which can be volatile. In this paper, we present a framework for testing volatility of P2P systems. The framework is based on the individual control of peers, allowing test cases to precisely control the volatility of peers during execution. We validated our framework through implementation and experimentation on two open-source P2P systems. Through experimentation, we analyze the behavior of both systems on different conditions of volatility and show how the framework is able to detect implementation problems.**

I. Introduction

A P2P system is composed of a volatile set of nodes, also called peers. Each peer can be a client and a server, as well as a router, since it can route incoming requests to other peers. Peers are autonomous, they can join and leave the system at any time, during the system lifetime. Such dynamic behavior distinguishes P2P systems from classical distributed systems. A P2P system must be able to work properly even though peers are highly volatile.

From the development point of view, a peer is an instance of a *P2P application*, which executes on a distinct logical node. Programming a peer is a difficult and error-prone task since it is part of a distributed system, with the classical synchronization issues, and it is programmed with various languages and platforms. P2P applications have two different interfaces, remote and local. The remote interface is used by other peers to request services through the network. The local interface is used by local applications to access the functionalities from the whole system. Samples of local applications are graphical user interfaces or applications that use the peer application as a middleware component.

A Distributed Hash Table (DHT) [1], [2] is an example of a P2P application, where each peer is responsible for the storage of values corresponding to a range of keys. It has a simple local interface that only provides three operations: value insertion, value retrieval and key lookup. The remote interface is more complex, providing operations for data transfer and maintenance of the routing table, i.e., the correspondence table between keys and peers, used to determine which peer is responsible for a given key. Testing these interfaces in a stable system is rather simple. However, testing that the routing table is correctly updated and that requests are correctly routed when peers leave and join the system requires a mechanism to simulate volatility and ensure that this simulation does not interfere with the test, i.e., that unanswered requests are not interpreted as faults. To our knowledge, no such mechanism exists to test system's functionality together with its robustness to nodes volatility.

Some approaches for P2P testing propose to randomly stop the execution of peers [3], [4], or to insert faults in the network [5]. While these approaches are useful to observe the behavior of the whole system, they are not totally adapted for testing a P2P application. Since they focus on the tolerance to network perturbations, they fail in detecting software faults, especially those which occurs due to peers volatility. For instance, if one wants to test if a peer is able to rebuild its routing table when all peers it knows leave the system, then the random departure of peers is not precise enough. Increasing significantly the percentage of volatile peers is not desirable either since the high number of messages necessary to rebuild the routing tables from the remaining peers may interfere with the test.

In this paper, we present a framework for testing P2P systems that can combine the execution of functional tests to the simulation of volatility. The capabilities of this framework are (1) to automate the execution of each local-to-a-peer test case, (2) to build automatically the global verdict, (3) to allow the explicit control of each peer volatility. This framework does not address the issue of test cases generation but is a first element towards an automated P2P testing process. It can be considered analogous to the JUnit[6] testing framework for Java unit tests. The framework is composed of two actors, the coordinator and the testers. Testers are applications that execute in the same logical node as peers and control their execution and their volatility, making peers leave and join the system at any time, according to the needs of a test. Thus, the volatility of peers can be controlled at a very precise level. The coordinator is an independent application that dispatches test case actions to testers and stores the state of each peer, avoiding the interference of unavailable peers with the execution of test cases.

Research partially supported by the Programme Alβan, the European Union Programme of High Level Scholarships for Latin America, scholarship no. E05D057478BR.

II. Testing P2P Systems

The classical architecture for testing a distributed system consists of a centralized tester which sends the test inputs, controls the synchronization of the distributed system and receives the outputs (or local verdicts) of each node of the system under test (SUT). In many cases, the distributed SUT is perceived as a single application and it is tested using its external functionalities, without considering its components (i.e., black-box testing). The tester in that case must interpret results which include non-determinism since several input/outputs orderings can be considered as correct.

Analyzing the specific features of P2P system, we remark that they are distributed systems, but the existing testing techniques for distributed systems do not address the issue of synchronization when a large number of nodes are involved. However, the typical centralized tester architecture can be reused for building a testing framework for P2P systems. Another issue that is not addressed by distributed system testing is the problem of node volatility. P2P systems must be robust and work even if peers are volatile (under limits which have to be determined). Volatility thus interferes with the system functionality and may cause failures to occur.

III. A Framework for Testing Peer Volatility

In this section, we present a framework for testing P2P systems. First, we define the basic concepts. Then, we describe the components of the framework. Finally, we explain how the framework executes test cases.

A. Basic Concepts

As described in the previous section, one particularity of a P2P system is that its interface is spread over a network. Thus, even if all peers have exactly the same interface, testing the interface of a single peer is not sufficient to test the whole system. For instance, consider a simple test case for a DHT, where the string "one" is inserted at key 1. Then, some data is retrieved at key 1. Finally, the retrieved value is compared with the string "one" to assign a verdict to the test case. Clearly, this test case does not ensure that all peers will be able to retrieve the data stored in key 1. This is why we introduce the notion of distributed test cases, i.e., test cases that apply to the whole system and whose actions may be executed by different peers.

Let us denote by P the set of peers representing the SUT, i.e., the P2P system. We denote by T, where $|T| = |P|$ the set of testers that controls the SUT, by DTS the suite of tests that verifies P, and by A the set of actions executed by DTS on P.

Definition 1 (Distributed test case): A distributed test case noted τ is a tuple $\tau = (A^\tau, T^\tau, L^\tau, S^\tau, V^\tau)$ where $A^\tau \subseteq A$ is an ordered set of actions $\{a_0^\tau, \ldots, a_m^\tau\}$, $T^\tau \subseteq T$ a set of testers, L^τ is a set of local verdicts, S^τ is a schedule and V^τ is a set of variables.

The Schedule is a map between actions and sets of testers, where each action corresponds to the set of testers that execute it.

Definition 2 (Schedule): A schedule is a map $S = A \mapsto \Pi$, where Π is a collection of tester sets $\Pi = \{T_0, \ldots, T_n\}$, and $\forall T_i \in \Pi : T_i \subseteq T$

In P2P systems, the autonomy and the heterogeneity of peers interfere directly in the execution of service requests. While close peers may answer quickly, distant or overloaded peers may need a considerable delay to answer. Consequently, clients do not expect to receive a complete result, but the available results that can be retrieved during a given time. Thus, test case actions must not wait indefinitely for results, but specify a maximum delay for an execution.

Definition 3 (Action): A test case action is a tuple $a_i^\tau = (\Psi, \iota, T')$ where Ψ is a set of instructions, ι is the interval of time in which Ψ should be executed and $T' \subseteq T$ is a set of testers that executes the action.

The instructions are typically calls to the peer application interface as well as any statement in the test case programming language.

Definition 4 (Local verdict): . A local verdict is given by comparing the expected result, noted E, with the result itself, noted R. E and R may be a single value or a set of values from any comparable type. The local verdict v of τ on ι is defined as follows:

$$l_\iota^\tau = \begin{cases} pass & if R = E \\ fail & if R \neq E \\ inconclusive & if R = \emptyset \end{cases}$$

B. Test Case Example

Let us illustrate these definitions with a simple distributed test case. The aim of this test case is to detect errors on a DHT implementation. More precisely, it verifies that new peers are able to retrieve data that was inserted before their arrival.

Example 1 (Simple test case):

Action	Testers	Instructions
(a_1)	0,1,2	join()
(a_2)	2	put(14, "fourteen");
(a_3)	3,4	join();
(a_4)	3,4	data := retrieve(14);
(a_5)	3,4	assert(data = "fourteen");
(a_6)	*	leave();

This test case involves five testers $T^\tau = \{t_0 \ldots t_4\}$ that control five peers $P = \{p_0 \ldots p_4\}$ and six actions $A^\tau = \{a_1^\tau, ..., a_6^\tau\}$. If the data retrieved in a_4 is the same as the one inserted in a_2, then the verdict is $pass$. If the data is not the same, the verdict is $fail$. If t_3 or t_4 are not able to retrieve any data, then the verdict is $inconclusive$.

C. Framework Components

The framework has two main components, the *tester* and the *coordinator*. The role of the tester is to execute test case actions and control the volatility of a single peer. The role of the coordinator is to dispatch the actions of a test case (A^τ) through the testers (T^τ) and maintain a list of unavailable peers. The UML diagram presented in Figure 1 illustrates the deployment of the framework: one coordinator controls several

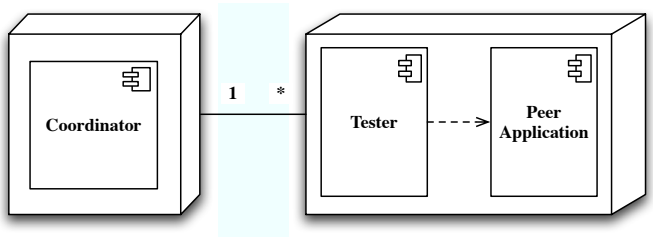

Fig. 1. Deployment Diagram

testers and each tester runs on a different logical node (the same as the peer it controls).

The tester component provides two interfaces, for action execution and volatility control:

1) **execute()**: executes a given action
2) **leave(), fail(), join()**: makes a set of peers leave the system, abnormally quit or join the system.

The coordinator component provides three different interfaces, for action execution, volatility and test case variables:

1) **register(), ok(), fail(), error()**: action registration (performed before all tests) and response for action execution, called by testers once the execution of an action is finished.
2) **set(), get()**: accessors for test case variables.
3) **leave(), fail(), join()**: makes a set of peers leave the system, abnormally quit or join the system.

D. Test Case Execution Algorithm

The algorithm has three steps: registration, action execution and verdict construction. Before the execution of a τ, each $t \in T$ register its actions with the *coordinator*. For instance, tester t_2 may register the actions $A' = \{a_1, a_2, a_6\}$. Once the registration is finished, the *coordinator* builds the schedule, mapping the actions with their related subset of testers. In our example, action a_3 is mapped to $\{t_3, t_4\}$.

Once S is built, the coordinator traverses all test cases $\tau \in DTS$ and then the actions of each τ. For each action a_i^τ, it uses $S(a_i^\tau)$ to find the set of testers that are related to it and then sends the asynchronous message $execute(a_i) \forall t \in S^\tau(a_i^\tau)$. Then, the coordinator waits for the available testers to inform the end of their execution. The set of available testers corresponds to $S^\tau(a) - T_u$, where T_u is the set of unavailable testers. In our example, once a_1 is finished, testers $\{t_0, t_1, t_2\}$ inform the *coordinator* of the end of the execution. Thus, the *coordinator* knows that a_1 is completed and the next action can start.

When a tester $t \in T^\tau$ receives the message $execute(a_n^\tau)$, it executes the suitable action. If the execution succeeds, then a message ok is sent to the coordinator. Otherwise, if the action timeout is reached, then a message $error$ is sent. Once the execution of τ finishes, the coordinator asks all testers for a local verdict. In the example, if t_3 gets the correct string "fourteen" in a_5, then its local verdict is $pass$. Otherwise, it is $fail$. After receiving all local verdicts, the coordinator is

Algorithm 1: Test suite execution

Input: T, a set of testers; DTS, a distributed test suite
Output: $Verdict$
foreach $t \in T$ **do**
  $register(t, A^t)$;
end
foreach $\tau \in DTS$ **do**
  foreach $a \in A^\tau$ **do**
    foreach $t \in S^\tau(a)$ **do**
      send execute(a) to t;
    end
    wait for an answer from all $t \in (S^\tau(a) - T_u)$;
  end
  foreach $t \in T^\tau$ **do**
    $L^\tau \leftarrow L^\tau + l_t^\tau$;
  end
  return $oracle(L^\tau, \varphi)$;
end

able to assign a verdict L^τ. If any local verdict is $fail$, then L^τ is also $fail$, otherwise the coordinator continues grouping each l_t^τ into L^τ. When L^τ is completed, it is analyzed to decide between verdicts $pass$ and $inconclusive$ as described in Algorithm 2. This algorithm has two inputs, a set of local verdicts (L) and an index of relaxation (φ), representing the level of acceptable $inconclusive$ verdicts. If the ratio between the number of $pass$ and the number of local verdicts is greater than φ, then the verdict is $pass$. Otherwise, the verdict is $inconclusive$.

Algorithm 2: Oracle

Input: L, a set of local verdicts; φ an index of relaxation
if $\exists l \in L, l = fail$ **then**
  return $fail$
else if $|\{l \in L : l = pass\}|/|L|) \geq \varphi$ **then**
  return $pass$
else
  return $inconclusive$
end

IV. Experimental Validation

In this section, we present an experimental validation of our framework. This is based on an experiment that verify the routing table structure of two popular open-source DHTs: (i) FreePastry[1]: an implementation of Pastry [7], from Rice University; and (ii) OpenChord[2]: an implementation of Chord [2], from Bamberg University.

The routing table is a common structure, used in most implementation of structured P2P systems. It stores the neighborhood of a peer, i.e., the peers it must know to maintain the system underlying structure and (indirectly) access the rest of the system. Chord and Pastry have different approaches to

[1]FreePastry: http://freepastry.rice.edu/FreePastry/
[2]OpenChord: http://open-chord.sourceforge.net/

update their routing table. While Chord uses an active process called stabilization, which periodically maintains the routing table, Pastry uses a lazy approach: the routing table is updated only when a peer communicates with its neighbors. In Chord, the components of a routing table are called *successors*.

For instance, in a Chord system with peers p_0, p_1, p_3, and p_6, the routing table of p_1 stores the addresses of its successors, p_3 and p_6. If a new peer p_4 joins the system, then p_1 will update its routing table with the address of p_4. The update of the routing table happens every time a peer joins or leaves the system.

For our experiment, we implemented our framework in Java (version 1.5), and we use a cluster of 64 nodes[3] running GNU/Linux. Each node has 2 Intel Xeon 2.33GHz dual-core processors with 4GB of memory. Since we can have full control over this cluster during experimentation, our experiments are reproducible. The implementation and tests, produced for this paper and other P2P applications, can be found in our web page.[4] Yet, we allocate the peers equally through the nodes in the cluster. In the experiment reported in this paper, each peer is configured to run in its own Java VM.

A. Recovery from Peer Isolation

In this experiment, we test the ability of a peer to update its routing table. This test consists in the departure of all peers that are present in the routing table of a given peer p and in the verification that this routing table is updated within a time limit. The test shows the ability of the framework to simulate and control the volatility of peers at a very precise level.

The test case has four actions. In the first one, the P2P system is created and a set of peers P joins the system. In the second action, a peer $p \in P$ (randomly chosen) stores the contents of its routing table in the test case variable RT. In the third action, the peers whose IDs are in RT leave the system. In the fourth action, the routing table of p is periodically analyzed within a delay. At the end of this action, the routing table of p is analyzed a last time to assign a verdict. The values of RT are compared with the updated routing table of p. If the intersection of these two sets is empty, the verdict is $pass$.

We created a system of 64 peers. Indeed, creating a system with less than 64 peers can lead the test to an $inconclusive$ result because p may know all the peers which are removed in the third action. In a larger system, the results would be similar since the update of the routing table is performed periodically. The test cases showed that both systems were able to update their routing table. While OpenChord updated its routing table in 4.32 sec., FreePastry needed 29.9 sec. A delay of 4 seconds was necessary to OpenChord to get a $pass$ verdict. This delay represents a unique execution of the stabilization process (whose periodicity is set to 6 seconds). In the case of FreePastry, two calls of a *ping* method are needed to force the update of the routing table and get a $pass$ verdict.

[3]The clusters are part of the Grid5000 project: http://www.grid5000.fr/

[4]Peerunit project, http://peerunit.gforge.inria.fr

V. Conclusion

In this paper, we proposed a framework for testing volatility of P2P systems. The framework is based on the individual control of peers, allowing test cases to precisely control the volatility of peers during tests. The execution of test cases is controlled by a coordinator, which ensures the correct synchronization of test case actions and avoid that volatility interferes with the execution of actions. The coordinator is also responsible for analyzing the results of test cases and for assigning their verdicts. We introduced the concept of distributed test cases, i.e., test cases that apply to several peers at the same time.

During the experiments, we focused on volatility testing and did not test these systems on more extreme situations such as performing massive inserts and retrieves or using very large data. Testing different aspects (concurrency, data transfer, etc.) would increase significantly the confidence on these systems. However, these tests are out of the scope of this paper. They could be performed through the interface of a single peer and would not need the framework presented in this paper.

Finally, the experiments exposed the need for a precise methodology for P2P testing, where the simplicity of interfaces contrasts with the complexity of the factors that can affect the test: volatility, number of peers, data size, amount of data, number of concurrent requests, etc. Thus, the difficulty of testing is not only in choosing the relevant input data, but also in choosing the factors that should vary, their values and their association.

References

[1] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenkern, "A scalable content-addressable network," *ACM SIGCOMM*, 2001.

[2] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," *ACM*, 2001.

[3] G. Antoniu, L. Bougé, M. Jan, and S. Monnett, "Large-scale deployment in P2P experiments using the JXTA distributed framework," JXTA - jdf.jxta.org, Tech. Rep., 2004.

[4] L. Yunhao, L. Xiaomei, X. Li, N. Lionel, and Z. Xiaodong, "Location-aware topology matching in P2P systems," *IEEE INFOCOM*, 2004.

[5] W. Hoarau, S. Tixeuil, and F. Vauchelles, "Fault injection in distributed java applications," in *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*, 2006.

[6] Junit, "http://www.junit.org," 2006.

[7] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, nov 2001, pp. 329–350.